EECS541 Mock Project Proposal

Stephen Longofono

August 2017

1 Introduction

The premise of this proposed capstone project is a generalized platform for tele-robotics. Using a web interface, a remote user or system will manage the operation of an arbitrary robot through a standardized set of high-level commands.

The platform itself will handle setting up and maintaining communication with the remote server, and translating high-level commands to specific lower level commands relevant to the hardware on hand. Wherever possible, the platform will streamline the process of configuring a new robot, especially with regard to internal communication and the transfer of data. New means of locomotion, sensing, or actuating are then a matter of writing a driver to the API of the high-level commands. In this way, any sort of motor system, sensors, or actuators can be supported with minimal additional code.

2 Problem Specification

This proposal will specifically address the system from the generalized interface down to the actual sensors and actuators. The web interface and any server-side data processing will be left to a server-side software team. To better define the problem at hand, the project is organized into areas of responsibility: System management, Sensor control, actuator control, drivetrain control, control API, and system setup.

System management is concerned with the operating system and how it manages the interaction of the sensors, actuators, drivetrain, and control system. It must be able to automatically configure itself on power-up, identify and initialize the appropriate programs to manage hardware, and manage a connection to the remote server. The latter may include connecting to a local server and using it as a getway to the web interface.

Sensor, actuator, and drive control is concerned with how to translate a general command from the operator to the specific low-level signals understood by the hardware on the robot at hand. Sensors and actuators may require intermediate software to implement protocols and pre-process data. Motor control may include software to realize pulse width modulation or to interact with a daughter board. Drivers for specific hardware are included in this area of responsibility.

The control API is concerned with providing a general and flexible interface to any robot; it is an abstraction of low-level tasks presented to the operator. Here, we must find a good tradeoff between ease of operation and fine-tuned control.

System setup is concerned with the process of registering specific configurations of hardware with the operating system. A manifest of sensors, actuators, and drivetrains and their relevant parameters must be captured in an organized and intelligent way, such that a user can more easily configure their robot by providing some specifications and minimal code. This section will also be responsible for linking the proper driver code to the control API.

3 Problem Constraints

The platform we select for the system-level tasks will need to be able to manage a WiFi connection, store sensor data, operate on battery power, and have both analog and digital GPIO pins. If possible, the platform should also be able to run multiple programming languages to accommodate a larger variety of drivers and intermediate software.

In software, both ends of the system will need to be automated: the system must be able to establish a connection to a remote server and respond to API commands in a timely manner. During the process of setting up a new robot, there must be a standard and simple means of registering new hardware and verifying its operation. The goal here is not completely automated setup, rather it is to make the process of configuring a robot easier for the end user.

These constraints are essential in creating a platform that is both accepting of a variety of hardware and one which improves the process of building a robot.

In order to limit the scope of the project to a reasonable level, it may be necessary to limit hardware support to a small selection of sensors, actuators, and drivetrains. A reasonable goal would be to support two each of digital and analog sensors, two drivetrain types (say, servo motor or propeller), and two actuator types (say, a relay and a stepper motor).

Additional constraints to consider include the ease of use for laypersons, and the flexibility to make our system useful across a wide variety of applications. Again, we must find a balance between how long it takes to set up our system and how flexible it is; if it is too simple, we may not provide enough control or detail to be useful in a scientific or industrial application. If we stray too far toward customization, we will lose the broad appeal that makes our system an improvement over existing robot control systems.

4 State of the Art and Similar Systems

Historically, robot and machine control systems have been custom-designed for the task at hand. Several efforts have been made to develop generalized robot control system for specific domains, which we examine here to inform our own work.

One of the most successful open-source systems is Orocos. Orocos is an EU-sponsored system for connecting disparate environments, components, and tasks into an automated control system. The advantages of Orocos are its extensive flexibility, its ability to handle real-time tasks, and its adherence to general inter-system communication schemes such as CORBA. Orocos uses its core module, the Real-Time Toolchain, to load, configure, and deploy components which represent processes or interactions. These components are generally compiled and configured in advance, and then loaded at runtime through a configuration manifest. Orocos also includes two sister projects, which focus on more advanced components to handle Bayesian filtering of sensor input and modeling kinematic/dynamic systems.

EEROS touts itself as the Easy, Elegant, Reliable, Open, and Safe robotics framework. EEROS architecture consists of three components: control, safety, and sequence. The control component handles individual tasks in terms of signals and time domains: input signals can be recorded via sensors, operated on by control blocks, and output as signals. Different time domains allow for periodic activities to easily synchronize with each other. The safety component continually monitors key inputs and outputs for pre-determined unsafe conditions, and acts accordingly to prevent damage to the system. The sequence component manages the synchronization and sequence of control blocks, and handles errors. EEROS is less flexible than Orocos, but is simpler and has better documentation.

Robot Operating System is a toolchain used to standardize the interaction, communication, and common tasks in robotics systems. ROS includes a number of common patterns for position estimation and sensor filtering, in addition to interfaces to common scientific tools such as opencv. The system was designed to run on multiple distinct nodes, and manage the communication between them via a publisher-subscriber communication pattern. ROS is a mature project with excellent documentation. It is used widely in academic, commerical, and industrial settings.

5 Design Choices

To meet our project goals in a timely manner, it would be prudent to select an existing robotics system as a baseline, and build our improvements as an abstraction over it. The complexity of writing and debugging an operating system would be time-prohibitive, and outside the scope of our project. By the same token, building our own custom ASIC or microcontroller is not feasible. In the interest of timely and reasonable development, we will use a microcomputer such as the Raspberry Pi or the Beaglebone. These systems will both run several distributions of Linux, and offer a wide variety of I/O devices and connectivity. Additionally, they have significant on-board storage to hold configuration information, buffered data, and system metadata.

Of the three systems described above in the State of the Art section, ROS is the best fit for our application. Orocos offers extensive flexibility and sophisticated toolchains, so much so that it may be too difficult to simplify using it without developing a significant amount of code. That is, in order to stub out the all the available possibilities into reasonable defaults, we would need to study and learn each part, decide if it should be exposed to the user, and if so write an abstraction of it into our own system.

EEROS and ROS both have the advantage of highly modularized components, simple architecture, and minimal setup for simple robots. They both fit our goals regarding connectivity, communication, and extensibility well. However, after a closer look at EEROS, it is still very much a work in progress. There are only a handful of examples beyond their demonstration robot to work from, and their community is still small. In contrast, ROS has hundreds of examples, support for a variety of common sensors, a huge community, and seven major versions in production. From an engineering standpoint, ROS has proven itself to be stable and useful, and EEROS has yet to do so. For these reasons, we will use ROS as our underlying robotic control system.

Using ROS as a starting point, there is still the problem of how to handle sensors, motors, and actuators which are not directly supported by ROS. Rather than write drivers for the thousands of components available, we will use a common signature for driver functions, which will be run in their own thread. This will allow users to download example drivers for supported sensors, and write their own if no driver yet exists.

As a part of the setup process, a hardware manifest (what hardware is available) will be used to locate the appropriate driver files, which can then be parsed to determine what functionality is available to the end user. This approach allows us to expose only the functions that the specific robot is capable of. There may also be the need for customized parameters in sensor or motor interaction; In the interest of streamlining setup, we will use a set of reasonable defaults, and allow the end-user to over-ride them with special commands.

To make our product useful to the widest variety of people, we will still need to provide the option of customization. The ability to write custom drivers works toward that goal, but we will also need to take care to make sure our architecture will be easily modified to support a variety of use cases. As a group, we have decided to make our project open source. Among other personal reasons, this will promote community involvement, allowing users with specialized applications to submit their own amendments to our core architecture. So long as we write our controlling software in a uniform and modular way, we will be able to provide the basic functionality that most need while still allowing for customization.

Exposing a generalized interface to a robot will require middleware to translate commands to a format that hardware drivers can use. We have decided that this responsibility is better kept on the user side, as it will vary depending on the application at hand. In case there is a need to group low-level commands together into a more general action, we will include support for scripts of commands.

There are many more design decisions to be made once we begin building and testing software, but the above are key first steps. As a follow-up to this proposal, we intend to field test ROS, our driver scheme, and general message passing to refine our decisions. Again, we will focus on streamlining the setup process and striking a balance between setup work and flexibility.

6 Project Viability

The proposed project is designed to streamline the process of setting up a robot, and thus its value depends on how well we are able to improve upon existing platforms. All of our decisions will be guided by this goal, but as discussed above, the scope of the task is potentially huge.

However, that scope is exactly why our project is necessary, and why any working abstraction will be an improvement. Industry and academia have encouraged highly specialized control systems far beyond what a hobbyist or small business might need. The investment of time and money into developing a custom robot is not feasible, and there are no products available to resolve this discrepancy.

Our project will provide a free and customizable alternative, with a wide variety of applications. The ability to use whatever components are available saves the end-user the cost of buying specialty hardware. The ability to reconfigure and add support for new parts makes our project useful as business needs change. As our community of developers and library of support drivers grows, so too will the utility of our system. In sum, there is a definite need for a product like ours and we believe it will be immediately useful to a large market.